

Matrix-free GPU implementation of a preconditioned conjugate gradient solver for anisotropic elliptic PDEs

Eike Müller^{*,1}, Xu Guo², Robert Scheichl¹ and Sinan Shi^{2,3}

¹*Department of Mathematical Sciences, University of Bath, Bath BA2 7AY, United Kingdom*

²*Edinburgh Parallel Computing Centre (EPCC), The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh EH9 3JZ, United Kingdom*

³*Current address: OT-Med, Europôle Méditerranéen de l'Arbois, Bâtiment du Cerege, BP 80 13545 Aix-en-Provence Cedex 4, France*

**Email: e.mueller@bath.ac.uk*

March 1, 2013

Abstract

Many problems in geophysical and atmospheric modelling require the fast solution of elliptic partial differential equations (PDEs) in “flat” three dimensional geometries. In particular, an anisotropic elliptic PDE for the pressure correction has to be solved at every time step in the dynamical core of many numerical weather prediction (NWP) models, and equations of a very similar structure arise in global ocean models, subsurface flow simulations and gas and oil reservoir modelling. The elliptic solve is often the bottleneck of the forecast, and to meet operational requirements an algorithmically optimal method has to be used and implemented efficiently. Graphics Processing Units (GPUs) have been shown to be highly efficient (both in terms of absolute performance and power consumption) for a wide range of applications in scientific computing, and recently iterative solvers have been parallelised on these architectures. In this article we describe the GPU implementation and optimisation of a Preconditioned Conjugate Gradient (PCG) algorithm for the solution of a three dimensional anisotropic elliptic PDE for the pressure correction in NWP. Our implementation exploits the strong vertical anisotropy of the elliptic operator in the construction of a suitable preconditioner. As the algorithm is memory bound, performance can be improved significantly by reducing the amount of global memory access. We achieve this by using a matrix-free implementation which does not require explicit storage of the matrix and instead recalculates the local stencil. Global memory access can also be reduced by rewriting the PCG algorithm using loop fusion and we show that this further reduces the runtime on the GPU. We demonstrate the performance of our matrix-free GPU code by comparing it both to a sequential CPU implementation and to a matrix-explicit GPU code which uses existing CUDA libraries. The absolute performance of the algorithm for different problem sizes is quantified in terms of floating point throughput and global memory bandwidth.

1 Introduction

Anisotropic elliptic PDEs arise in many areas of geophysical and atmospheric modelling, which are often characterised by “flat” geometries: the horizontal extent of the domain of interest is much larger than its vertical size. This is the case for global weather- and climate prediction models. As the height of the atmosphere is significantly smaller than the radius of the earth, the horizontal resolution is of the order of 10-25 kilometers but the vertical grid spacing is several tens or hundreds of metres. Similar ranges of scales are encountered in models for simulating global ocean currents. Due to the layered structure of geological formations, oil and gas reservoir simulations and subsurface flow models of aquifers are also typically carried out in “flat” domains. After discretisation the cells of the computational grid are very flat and the resulting matrix stencil is highly anisotropic, i.e. the coupling in the vertical direction exceeds the horizontal

coupling by several orders of magnitude. To achieve optimal performance, it is important to exploit the strong anisotropy of the system when choosing an appropriate computational grid and an efficient solver.

In this work we focus on the elliptic PDE for the pressure correction arising in the dynamical core of numerical weather- and climate- prediction models. In many forecast models semi-implicit semi-Lagrangian time stepping introduced in [1] and [2] is used to advance the atmospheric fields forward in time. In contrast to explicit time stepping this method has a larger stability region and allows for longer model time steps without compromising the accuracy of the large scale dynamics, which can reduce the overall model runtime. However, if this approach is used to solve the fully compressible non-hydrostatic Euler equations, a three dimensional PDE for the pressure correction has to be solved at every time step as discussed for example in [3, 4, 5, 6, 7, 8],

which often forms the computationally most expensive proportion of the model runtime.

Algorithmically the most efficient solvers for large elliptic PDEs are suitably preconditioned Krylov subspace- or multigrid methods (see e.g. [9, 10, 11, 12]). The strong anisotropy in the vertical direction can be exploited to construct an efficient preconditioner (or multigrid smoother) based on vertical line relaxation as discussed in [4, 5]. In an related context [13] and [14] describe how the equations of ocean flows can be discretised on a tensor product grid which is unstructured in the horizontal but consists of regular columns in the vertical direction. Again the strong anisotropy is used in the construction of an efficient preconditioner of the iterative solver. In a similar fashion anisotropic elliptic PDEs arise in fully implicit methods for gas- and oil reservoir modelling. A “supercoarsening” multigrid algorithm for solving elliptic PDEs encountered in multiphase flow in porous media is described by [15]: while the full three dimensional equation is solved on the finest grid, any vertical variations are averaged out on the coarser multigrid levels by collapsing vertical columns to a single layer.

The exact hardware on which forecast models will be implemented in the future is currently unknown, and it is important to explore novel chip architectures in addition to standard CPUs. Graphics Processing Units (GPUs) are fast and power- efficient computing devices and significant speedups relative to standard CPU implementations have been achieved in the past for iterative solvers for elliptic PDE, as described in [16, 17, 18, 19, 20, 21, 22, 23, 24, 25].

While modern multicore CPUs contain several tens of cores and have a peak floating point performance of $\mathcal{O}(10 - 100)$ GFLOPs, GPUs have several hundreds to thousands of cores and applications have to make efficient use of the massively parallel SIMD architecture and limited cache size per thread. The nVidia M2090 Fermi GPU, on which this work was carried out, has a peak performance of 1.331/0.665 TFLOP/s in single and double precision respectively and a global memory bandwidth of 177GByte/s. By dividing the peak FLOP rate by the memory bandwidth on the GPU, one can deduce that the number of computations per floating point variable read from memory is around 30, so computations are essentially “free” and the performance is limited by the speed with which data can be read from global memory and how efficiently it can be kept in cache.

In this article we describe a matrix-free GPU implementation of a preconditioned Conjugate Gradient (PCG) solver tailored towards the solution of anisotropic PDEs with a tensor-product structure. The most compute intensive components of the iterative solver are the evaluation of a large sparse matrix-vector product (SpMV) and the inversion of a block-tridiagonal matrix. Both kernels were ported to the GPU and the memory access pattern and thread layout were adapted to increase data throughput. In our implementation the matrix is not stored explicitly but recalculated in every grid cell, which reduces access to global memory compared to the matrix-explicit code. However the stencil we

use is more complicated than the simple Poisson stencil on a regular grid used in previous studies (see [17, 19]). Due to the tensor product structure of the equation and the computational grid, the matrix can be written as the product of a one dimensional vertical discretisation, which is the same for every column and only needs to be calculated and stored once, and a horizontal stencil. As the number of vertical levels is very large in atmospheric applications, and the horizontal coupling only needs to be calculated once for each vertical column, this creates only a small overhead. The vertical discretisation requires the storage of four vectors of length n_z , where n_z is the number of vertical levels. In total we store $4 \times n_z$ values to parametrise the matrix. With the exception of [17, 19], all implementations discussed in the literature that we are aware of, store the matrix explicitly, which requires the storage of $7 \times n_{\text{horiz}} \times n_z$ matrix entries, where n_{horiz} is the number of horizontal grid cells. This can have a negative impact on the performance on bandwidth limited architectures as it requires reading the matrix stencil from global memory in addition to the field vectors. While the specific implementation described in this article is based on a three dimensional grid which can be written as the tensor product of regular horizontal grid and a graded vertical grid, the method we present can also be applied to unstructured horizontal grids.

As the sparse matrix-vector product and preconditioner solve are highly efficient in our GPU implementation, other parts of the main CG such as level 1 BLAS vector updates and scalar products start to account for a significant proportion of the runtime, even if they are implemented using optimised GPU libraries such as CUBLAS. We find that to achieve further performance increases it is not sufficient to optimise the kernels in isolation, but rather several components of the main CG iteration need to be considered together, as has been suggested in [20]. In particular we find that the number of memory references can be reduced further by fusing the loops over the computational grid in the main kernels with the BLAS operations and this can lead to an additional performance gain of around 30%. For the entire PCG algorithm we are able to obtain a total speedup of a factor $60\times$ for single precision arithmetic on a nVidia Fermi M2090 card relative to one core of an Intel Xeon Sandybridge E5-2620 CPU. For double precision arithmetic the speedup was slightly smaller with $48\times$. This includes time for setting up the discretisation and copying data between host and device. To study the performance of our matrix-free GPU code we compared it to an implementation which stores the matrix explicitly in the compressed sparse row storage (CSR) format using the CUSPARSE and CUBLAS libraries. Our matrix-free code is significantly faster than the implementation based on CRS data structures which does not exploit the regular structure of the problem. We quantified the absolute performance in terms of floating point operations per second (FLOPs) and global memory bandwidth for different problem sizes, where the latter is the more relevant measure for the performance

of a memory bound algorithm. The optimised matrix-free code achieves a bandwidth of around 25% – 50% of the theoretical peak value, which is a sizeable proportion but shows that theoretically there is still potential for additional improvements which could lead to a further speedup of a factor $2 \times -4 \times$. An idea of how this could be achieved by increasing the granularity of the algorithm is discussed below. The floating point performance is 70-80 GFLOPs for single precision and 40-50 GFLOPs for double precision, corresponding to around 5–8% of the theoretical peak value.

Overview. This article is organised as follows: previous GPU implementations of iterative solvers for PDEs are reviewed in section 2. In section 3 the model equation and its discretisation is described in detail with particular emphasis on the tensor-product structure of the grid and the elliptic operator. Preconditioned Krylov-subspace solvers and the matrix-free and interleaved form of the PCG algorithm for solving the model equation are presented in section 4. A general overview over the GPU architecture and the CUDA programming- and execution model can be found in section 5, and our CUDA implementation of the PCG solver is described in section 6. Performance measurements are discussed in section 7 where we also present comparisons to a matrix-explicit implementation and quantify the absolute performance. Our conclusions and a discussion of planned further work can be found in section 8. For reference appendix A contains the explicit form of the two most important kernels of our optimised algorithm.

2 Previous work

The GPU implementation of Krylov-subspace solvers and in particular of the Preconditioned Conjugate Gradient algorithm has been studied extensively in the literature, both for more general sparse matrices and for matrices arising from the discretisation of elliptic PDEs. As far as we are aware, all implementations discussed in the literature (with the exception of [19] and [17]) are based on matrix-explicit representations. While some of the authors study Poisson- or sign-positive Helmholtz equations, none of the problems studied in the literature show the strong anisotropy which characterises the elliptic operator we consider in this work, and hence the preconditioners investigated in the literature will not be optimal in our case.

While the speedups presented in the following review depend on the problem and on the hardware used at a particular time and should only be used as an indicator for achievable performance gains, almost all GPU implementations are significantly faster than the corresponding CPU versions with speedups of $20 \times -40 \times$ relative to the sequential code.

Some early work is presented in [16] where both a conjugate gradient and a multigrid solver are implemented for

solving the sign positive Helmholtz equation $-\Delta u + \sigma u = g$ arising from an implicit time discretisation of the incompressible Navier-Stokes equations on a regular two dimensional grid. However, for both solvers the matrix is stored explicitly.

The compressed sparse row storage format (CSR) is a very popular and general format which has been used in a variety of recent GPU implementations of Krylov subspace algorithms. A PCG solver for the same sign-positive Helmholtz equation as in [16] is described in [23] for two and three dimensions. For the approximate inverse SSOR preconditioner, which requires an additional matrix multiplication, a socket-to-socket speedup of more than $8 \times$ is reported for the best implementation of the PCG algorithm on an nVidia Tesla T10 card relative to the unpreconditioned CG algorithm on an Intel Xeon Quad-Core 2.66 GHz CPU. Although in contrast to [16] a three dimensional system is solved, the elliptic operator considered is fully isotropic in both cases. In [20] a modified version of the Conjugate Gradient algorithm is used for solving a set of general matrices from the University of Florida sparse matrix collection described in [26]. A simple Jacobi preconditioner is used and the performance of the solver is optimised by using the prefetch CSR sparse matrix-vector multiplication in [27] and fusing kernels in the main PCG loop. As described in section 4.3 below we use a similar technique for fusing different kernels in our implementation to improve the performance of the code. Together with some other improvements the authors of [20] report that this led to a significant speedup compared to a GPU implementation using the “Row per warp” sparse matrix-vector multiplication described in [28]. One of the problems studied in [20] is the “thermal2” matrix which arises from an FEM discretisation of the stationary heat equation $\partial_x(k\partial_x T) + \partial_y(k\partial_y T) = 0$ on an unstructured two dimensional grid. For this problem a speedup of $41 \times$ relative to a single core of a Intel Core2 2.4GHz could be achieved on both nVidia GT8800 and GTX280 GPUs. The GPU implementation of a Krylov subspace solver for the (sign-indefinite) two dimensional Helmholtz equation is described in [21]. A shifted Laplace multigrid preconditioner is used to reduce the number of iterations and a speedup of around $30 \times$ could be achieved on an nVidia GeForce 9800 GTX/9800 GTX+ GPU relative to the sequential implementation on one core of an AMD Phenom 9850 CPU.

Other sparse matrix storage formats have also been used to implement iterative solvers on GPUs. An implementation of a CG solver for the two dimensional PDE arising from the implicit time discretisation of the heat equation is described in [29]. The ELLPACK-R data format, which is more suitable for structured problems, was used for storing the matrix and a speedup of a factor $26 \times$ could be achieved for a two dimensional problem of size 2048×2048 on an nVidia GeForce GTX 480 card, relative to the sequential implementation on an Intel Core i7 860 CPU with 2.80GHz. Although the structure of the five point nearest-neighbour stencil arising from a finite-difference approximation of the Poisson equation

tion is similar to the stencil we use in our discretisation, in contrast to our problem the elliptic PDE solved in [29] is two dimensional and fully isotropic. The GPU implementation of preconditioned GMRES and Conjugate Gradient solvers for a range of problems and preconditioners has been studied in [25] and the performance for different sparse matrix storage formats is compared. While in some of the implementations only the sparse-matrix vector product is carried out on the device and the preconditioner is executed on the host, preconditioners that are easier to parallelise are also ported to the GPU. However, the authors find that for a simple block-Jacobi preconditioner implemented on the GPU the number of iterations is very large. This should be compared to our implementation: for the strongly anisotropic elliptic PDE we consider the blocks have a direct physical interpretation as they described the strong vertical coupling within one column, which is much larger than the coupling between different blocks. As a result, the simple block-diagonal preconditioner proved to be very efficient in our numerical tests. In [22] the GPU implementation of a Preconditioned Conjugate Gradient solver for both a two dimensional Poisson problem and the elliptic equation arising in the Variational Boussinesq Model (VBM) is described. A Repeated Red Black (RRB) preconditioner is used, but an incomplete Poisson preconditioner with diagonal scaling is also considered. As for the sparse approximate inverse used in [23], the incomplete Poisson preconditioner can be reduced to an additional sparse matrix product. The sparse matrix-vector product is implemented by storing the local five-point stencil at each gridpoint. For the RRB preconditioner a speedup of around $40\times$ could be achieved for the Poisson test problem on an nVidia GeForce GTX 580 card, relative to the sequential implementation on an Intel Xeon W3520 CPU. However, the costs for memory allocation and setup of the preconditioner matrix take up around a third of the total runtime. In contrast in our matrix free implementation only a small amount of data has to be copied between host and device and the matrix setup costs are negligible. For realistic problems the speedup reported in [22] is $20\times$ – $30\times$ for the RRB preconditioner and $5\times$ – $20\times$ for the incomplete Poisson preconditioner.

As far as we are aware, the only matrix-free implementation of a CG solver discussed in the literature are [17, 19]. In [17] both the homogenous Poisson equation and the Navier Stokes equation are solved on an unstructured mesh by implementing matrix-free gradient and divergence operations. On an nVidia a speedup of around $3\times$ was achieved on an nVidia 6600GT card relative to an AMD Athlon 64 CPU. Note, however, that the implementation is based on the low-level graphics API and the hardware used in the study is quite dated by current standards. The GPU implementation of a matrix-free PCG solver for the homogenous Poisson equation in three dimensions is also described in [19].

Due to its significance in many scientific applications and in particular iterative solvers, the performance of sparse matrix-vector multiplications on its own has been studied

extensively in the literature: Various sparse matrix formats are described in [28] and their performance for a sparse matrix-vector multiplication is compared for both structured and unstructured matrices. While CSR is the most general format and can be applied to matrices with widely varying row sizes, the best performance for structured matrices arising from the discretisation of PDEs is obtained with the DIA and ELLPACK formats. However, in [24] an efficient parameter dependent implementation of sparse matrix-vector multiplication based on the CSR format on cache based GPUs is described. Cache usage and performance can be improved significantly by varying the number of threads processing each row, the thread block size and number of rows processed by one cooperating thread group. The authors find that by tuning the parameters heuristically, the performance of a Conjugate Gradient solver for a structured problem arising from the finite element discretisation of a simple elliptic Poisson problem using CSR storage is comparable to the corresponding implementation using the ELLPACK format. Both matrix-explicit versions are beaten by an implementation which stores a small local matrix on each element and assembles the global stiffness matrix on-the-fly in each matrix-vector product as described in [30, 31]. For other work on improved matrix-explicit implementations of the sparse matrix-vector product see the review and references cited in [24].

The number of iterations can often be reduced significantly by using multigrid methods, and recently both geometric and algebraic multigrid solvers have been ported to GPUs, see e.g. [32, 33, 34]. The extension of our PCG solver to a geometric multigrid solver for anisotropic problems based on the tensor product idea in [35] is discussed in [36] and we are currently working on a GPU implementation of the same matrix-free geometric multigrid solver.

We finally remark that multiple-GPU implementations of iterative solvers have been described in the literature (see e.g. [37, 38, 19, 39]) and we are currently working on extending our algorithm to clusters of GPU.

3 Model equation

Following [7, 8] a model equation which reproduces the most important features of the full PDE for the pressure correction in a NWP model, has been derived in [36]. The derivation of the full pressure correction equation in atmospheric models can be found for example in [3, 4, 5, 6] and is described for ocean models in [13] and [14]. The model equation we use is a symmetric positive definite PDE and can be written in spherical coordinates as

$$-\omega^2 \left(\Delta_{2d} + \lambda^2 \frac{1}{r^2} \frac{\partial}{\partial r} \left(r^2 \frac{\partial}{\partial r} \right) \right) u + u = f \quad (1)$$

where Δ_{2d} denotes the two dimensional Laplacian on the unit sphere. For simplicity all length scales are measured in units of the earth radius R_{earth} , and the equation is solved in a thin spherical shell $r \in [1, 1 + H_{\text{atmos}}]$. We write

$H_{\text{atmos}} = D/R_{\text{earth}} \ll 1$ where D is the thickness of the atmosphere. The model parameters ω^2 and λ^2 depend on the model time step size, in particular, ω is proportional to the time step size. In our numerical experiments the parameters were adjusted to their values for typical model resolutions in NWP with a Courant number of around 10. An important feature of the discretised PDE is the strong anisotropy in the vertical direction: the depth of the atmosphere is about two orders of magnitude smaller than the radius of the earth and consequently the horizontal grid spacing Δx is significantly larger than the vertical grid spacing Δz . The strong grid aligned anisotropy in the vertical direction is given (approximately) by

$$\gamma^2 = \left(\frac{\lambda \Delta x}{\Delta z} \right)^2, \quad (2)$$

and as $H_{\text{atmos}} \ll 1$ we have $\Delta z \ll \Delta x$, so $\gamma^2 \gg 1$. This property can be used to construct a simple but very efficient and parallelisable preconditioner for iterative solvers of this equation, which solves the vertical equation exactly but ignores the horizontal couplings.

The condition number of the preconditioned operator approaches a fixed value as the horizontal resolution increases and does not diverge as for the Poisson equation. To see this, note that to keep the Courant number constant, the time step size $\Delta t \propto \omega$ has to be chosen to be proportional to the horizontal grid spacing, i.e. $\Delta t \propto \Delta x$. The largest eigenvalue of the preconditioned matrix is of the order $\omega^2/\Delta x^2 \propto \Delta t^2/\Delta x^2$ and independent of Δx , whereas the smallest eigenvalue is 1 due to the presence of the zero order term in (1).

After discretisation the elliptic operator in (1) can be written abstractly in tensor product form as the sum of three terms:

$$L = D^{(h)} \otimes M^{(r)} + M^{(h)} \otimes D^{(r)} + \tilde{M}^{(h)} \otimes \tilde{M}^{(r)} \quad (3)$$

Here $M^{(h)}$ and $\tilde{M}^{(h)}$ ($M^{(r)}$ and $\tilde{M}^{(r)}$) are horizontal (vertical) mass matrices which only contain couplings in the horizontal (vertical) direction. $D^{(h)}$ ($D^{(r)}$) are second order derivative operators which contain couplings in the horizontal (vertical) direction.

3.1 Discretisation

To discretise (1) we use a finite volume scheme on a tensor-product grid, which consists of a (possibly unstructured but conforming) two dimensional grid on the unit sphere and a non-uniform (typically graded) one dimensional grid in the vertical direction. The model fields are defined as integrals in a grid cell given by the horizontal grid element T and vertical index $k = 0, \dots, n_z - 1$

$$\bar{u}_k^{(T)} \equiv \int_{r_k}^{r_{k+1}} \int_T u(r, \boldsymbol{\theta}) r^2 dr d\boldsymbol{\theta} \quad (4)$$

with $\boldsymbol{\theta}$ denoting horizontal coordinates on the unit sphere (throughout this work we use zero-based indexing as all our

implementations are in the C programming language). Independent of the horizontal discretisation we need to store a vector $\bar{\mathbf{u}}^{(T)}$ of length n_z at each element T . The vertical grid is defined by the grid points r_k where $k = 0, \dots, n_z$. The number of vertical grid cells is usually large, $n_z = \mathcal{O}(100)$. In meteorological applications a graded vertical grid with smaller grid spacings near the ground is desirable and we use $r_k = 1 + (k/n_z)^2 \cdot H_{\text{atmos}}$.

Equation (1) is discretised using a finite volume scheme, and schematically it can be written for each horizontal grid element T as

$$(\mathbf{A}\bar{\mathbf{u}})^{(T)} = \mathbf{A}_T \bar{\mathbf{u}}^{(T)} + \sum_{T' \in \mathcal{N}(T)} \mathbf{A}_{T,T'} \bar{\mathbf{u}}^{(T')} = \bar{\mathbf{f}}^{(T)} \quad (5)$$

where $\mathcal{N}(T)$ is the set of neighbours of T . \mathbf{A}_T is a tridiagonal $n_z \times n_z$ matrix and $\mathbf{A}_{T,T'}$ are diagonal $n_z \times n_z$ matrices for each neighbouring element T' . The matrices $\mathbf{A}_{T,T'}$ correspond to the off-diagonal couplings in the horizontal derivative operator $D^{(h)} \otimes M^{(r)}$ in (3) and are given by the product

$$\mathbf{A}_{T,T'} = \alpha_{T,T'} \text{diag}(\mathbf{d}) = \alpha_{T,T'} \begin{pmatrix} d_0 & & & \\ & d_1 & & \\ & & \ddots & \\ & & & d_{n_z-1} \end{pmatrix}. \quad (6)$$

The coefficient $\alpha_{T,T'}$ is the ratio between the length $S_{T,T'}$ of the edge between the cells T and T' and the distance between the midpoints \mathbf{r}_T and $\mathbf{r}_{T'}$ of these cells. We also define the diagonal coefficient α_T as the sum of the off-diagonal coefficients

$$\alpha_T \equiv \sum_{T' \in \mathcal{N}(T)} \alpha_{T,T'}. \quad (7)$$

The (symmetric) tridiagonal matrix \mathbf{A}_T can be split into a sum of three terms:

$$\begin{aligned} \mathbf{A}_T &= \begin{pmatrix} \tilde{d}_0 & b_0 & & \\ c_1 & \ddots & \ddots & \\ & \ddots & \ddots & b_{n_z-2} \\ & & c_{n_z-1} & \tilde{d}_{n_z-1} \end{pmatrix} \\ &= |T| \text{diag}(\mathbf{a}) - \alpha_T \text{diag}(\mathbf{d}) \\ &\quad + |T| \text{tridiag}(-(\mathbf{b} + \mathbf{c}), \mathbf{b}, \mathbf{c}), \end{aligned} \quad (8)$$

with $\tilde{d}_k = |T|(a_k - b_k - c_k) - \alpha_{ij} d_k$ where $|T|$ is the area of the horizontal grid element T . The first term corresponds to the product $\tilde{M}^{(h)} \otimes \tilde{M}^{(r)}$ in (3). The second term is the diagonal contribution of $D^{(h)} \otimes M^{(r)}$, and the third term corresponds to the vertical derivative term $M^{(h)} \otimes D^{(r)}$. While the finite volume discretisation described so far leads to a seven-point nearest-neighbour stencil on a regular grid, the same structure also arises for other stencil types which include couplings between grid cells which are not directly adjacent.

The vectors \mathbf{a} , \mathbf{b} , \mathbf{c} and \mathbf{d} do not depend on the grid cell T and can be precomputed. On the other hand the quantities

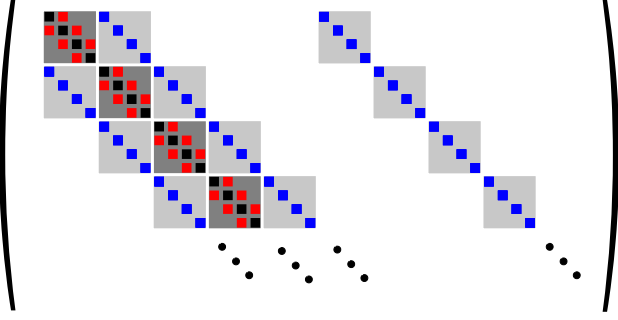


Figure 1: Structure of the matrix arising from the finite volume discretisation of (1) for $n_z = 4$ vertical columns. Vertical couplings are shown in red and horizontal couplings in blue.

$|T|$ and $\alpha_{T,T'}$ only need to be calculated once per vertical column. These two important observations will be exploited in the efficient matrix-free implementation of the PCG solver described below.

3.2 Global matrix representation

Assuming an ordering of the horizontal degrees of freedom, which maps each cell T to a linear index $\nu(T) \in 0, \dots, n_{\text{horiz}} - 1$, one can write the full 3d solution vector of length $n = n_{\text{horiz}} \times n_z$ as

$$\mathbf{u} = \left\{ \bar{\mathbf{u}}^{(0)}, \bar{\mathbf{u}}^{(1)}, \dots, \bar{\mathbf{u}}^{(n_{\text{horiz}})} \right\} \quad (9)$$

with

$$u_\ell = \bar{u}_k^{(T)} \quad \text{where} \quad \ell = n_z \cdot \nu(T) + k. \quad (10)$$

In this representation the discretised equation (5) can be written in the familiar form as

$$\mathbf{A}\mathbf{u} = \mathbf{f} \quad (11)$$

and the structure of the matrix \mathbf{A} is shown in Fig. 1. Note that the matrix has a block structure, where each of the blocks corresponds to one combination (T, T') of neighbouring elements of the horizontal grid. Each of the gray blocks is of size $n_z \times n_z$, the dark gray blocks ($T' = T$) describe the diagonal terms and vertical coupling, whereas the light gray blocks ($T \neq T' \in \mathcal{N}(T)$) describe the horizontal coupling. In the following we work on one panel of a cubed sphere grid with gnomonic projection (Fig. 2). This defines a mapping from $\tilde{\Omega} = [-1, 1] \times [-1, 1]$ to spherical coordinates on one-sixth of the entire sphere and each cell of the regular grid of size $n_{\text{horiz}} = m \times m$ on $\tilde{\Omega}$ can be labelled with an index pair $(i, j) \in [0, m-1] \times [0, m-1]$. In this case we have

$$u_\ell = \bar{u}_k^{(i,j)} \quad \text{where} \quad \ell = n_z (m \cdot i + j) + k \quad (12)$$

and label each grid cell T_{ij} by its horizontal indices. We stress, however, that there is no algorithmic problem in extending our approach to unstructured horizontal grids or to the entire spherical shell for example by ordering the horizontal grid cells along a space-filling curve.

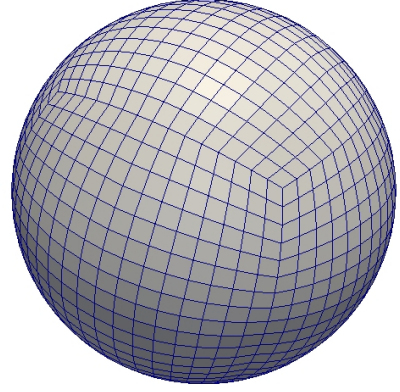


Figure 2: Cubed sphere grid. The model equation was discretised on one logically rectangular panel of this grid.

4 Iterative solvers for elliptic PDEs

Typically the number of degrees of freedom per atmospheric variable in current global forecast models is at the order of several 10 millions. For next generation forecast models global horizontal model resolutions of around one kilometre are envisaged, which will require the solution of PDEs with more than 10^{10} unknowns. Clearly naive direct methods can not be used for the solution of equations of this size and spectral methods often require a regular underlying grid structure and are hard to parallelise.

Krylov subspace methods (see e.g. [12]) are very efficient iterative algorithms for solving sparse linear systems, in particular if they are suitably preconditioned. These methods construct the approximation $\mathbf{u}^{(k)}$ of the exact solution \mathbf{u} of (11) in a k -dimensional Krylov-subspace

$$\mathcal{K}_k = \text{span} \left\{ \mathbf{r}, \mathbf{A}\mathbf{r}, \mathbf{A}^2\mathbf{r}, \dots, \mathbf{A}^{k-1}\mathbf{r} \right\} \subset \mathbb{R}^n, \quad (13)$$

where \mathbf{r} is the initial residual $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{u}^{(0)}$. They are easy to parallelise as in addition to local operations such as sparse matrix-vector multiplications and `axpy`-like vector updates they require only a small number of global reductions.

4.1 Preconditioned Conjugate Gradient algorithm

The simplest Krylov subspace method, which can be applied for symmetric positive definite matrices, is the Conjugate Gradient (CG) algorithm introduced in [9]. At each step the approximate solution vector $\mathbf{u}^{(k)}$ is updated by adding a correction proportional to the search direction $\mathbf{p}^{(k)}$. The search directions are chosen such that they are \mathbf{A} -orthogonal for different k, k' : $\langle \mathbf{p}^{(k)}, \mathbf{A}\mathbf{p}^{(k')} \rangle = 0$ for $k \neq k'$. The convergence rate depends on the spectral properties of the matrix \mathbf{A} , in particular on the condition number κ , which is the ratio between the largest and smallest eigenvalue, as derived in [12]. For the system arising from the discretisation of the Poisson equation, κ grows rapidly with the inverse

grid spacing $1/h$ and it can be shown that asymptotically for $h \rightarrow 0$ the number of iterations required to reduce the error by a factor ϵ is

$$k \propto \frac{\log \epsilon}{h}. \quad (14)$$

For anisotropic systems h is the smallest grid spacing in the problem, for the highly anisotropic problem in this work this is the vertical grid spacing $\Delta z \ll \Delta x$. The dominant cost in each iteration is the sparse matrix-vector multiplication $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$, which is of $\mathcal{O}(n)$ computational complexity. Hence the total cost of the algorithm is

$$\text{Cost}(\text{CG}) \propto \frac{n}{h} \log \epsilon. \quad (15)$$

To solve non-symmetric systems, more general Krylov space methods such as GMRES, BCG, BiCGStab and GCR can be used. Although the number of sparse matrix-vector products and intermediate vectors which need to be stored changes between different Krylov subspace algorithms, their general structure is very similar and in this work we focus on the Conjugate Gradient algorithm for simplicity.

An equivalent version of the linear system in (11) can be obtained by multiplication with the inverse of the matrix \mathbf{M} ,

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{u} = \mathbf{M}^{-1}\mathbf{f}. \quad (16)$$

This is generally referred to as left preconditioning. \mathbf{M} is a matrix which should be easy to invert (i.e. it should be easy to solve the system $\mathbf{M}\mathbf{x} = \mathbf{y}$) and as similar to \mathbf{A} as possible, such that the preconditioned matrix $\mathbf{M}^{-1}\mathbf{A}$ is well conditioned. Usually these two requirements are mutually exclusive and a tradeoff between them has to be found. Initial profiling of the CPU code revealed that the sparse matrix-vector multiplication and preconditioner solve account for the largest proportion of the runtime (80 – 90%). In addition to these operations each iteration requires three `axpy`-like vector updates, one `scal`-operation and two scalar products (`dot`). An additional norm calculation (`nrm`) is required for the evaluation of the stopping criterion. The number of floating points operations and memory references for the individual level 1 BLAS operations is given in Tab. 1, the total number of FLOPS per grid cell for all BLAS operations is 13 and the number of memory references is 16, and hence this part of the algorithm is clearly memory bound.

The strong anisotropy of the discretised PDE can be used to construct an efficient preconditioner: if the small horizontal couplings are ignored, the matrix \mathbf{A} shown in Fig. 1 is block-diagonal with each block corresponding to one vertical column. Each of the tridiagonal blocks can be inverted independently using the Thomas algorithm written down explicitly in [40]. More efficient block-SOR preconditioners can be used as well, and require the inversion of the same block-diagonal matrix plus an additional sparse matrix-vector product.

This approach has been applied very successfully for the pressure solver in the dynamical core of several numerical

operation	FLOPs	MEM
<code>scal</code>	1	2
<code>axpy</code>	2	3
<code>dot</code>	2	2
<code>nrm2</code>	2	1
total (PCG)	13	16

Table 1: Number of floating point operations and memory references per grid cell for different level 1 BLAS operations. The last row shows the total number of FLOPs and memory references for all BLAS operations in the PCG algorithm.

weather- and climate prediction models, see [41, 5, 6, 42]. In particular the authors of [4] show the efficiency of a simple 1d line relaxation in comparison to other preconditioners such as 2d ADI or a three dimensional pointwise SOR iteration. The good weak and strong scaling on up to 65536 cores of the HECToR Cray supercomputer and more than 10^{10} degrees of freedom is demonstrated for the model equation (1) in [36].

As discussed in section 3, the condition number of the preconditioned elliptic operator approaches a fixed value for physical choices of the parameter ω in (1) as the horizontal grid spacing tends to zero. Hence for these parameters Krylov subspace solvers for this equation are algorithmically stable in the sense that the number of iterations does not diverge as the horizontal resolution increases. However, as shown in [36] the number of iterations and total solution time can be reduced significantly by using (geometric) multigrid solvers.

4.2 Matrix-free implementation

Neither the matrix \mathbf{A} nor the preconditioner matrix \mathbf{M} need to be stored explicitly in the algorithm, it is sufficient to evaluate the sparse matrix-vector product $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$ and to solve the equation $\mathbf{M}\mathbf{x} = \mathbf{y}$ for \mathbf{x} . For matrices arising from the discretisation of PDEs, such as the one discussed in section 3, the local matrix stencil only couples each grid cell to its neighbours. As memory access is significantly more expensive than floating point operations on GPUs, it will be beneficial to recalculate the stencil whenever it is needed in the sparse matrix-vector product or preconditioner solve. In each horizontal grid cell (i, j) the diagonal matrices $\mathbf{A}_{T_{ij}, T_{i'j'}}$ in (6) (with $T_{i'j'} \in \mathcal{N}(T_{ij})$) and the tridiagonal matrix $\mathbf{A}_{T_{ij}}$ in (8) are calculated from the precomputed vectors \mathbf{a} , \mathbf{b} , \mathbf{c} and \mathbf{d} . For efficiency, we instead store the vectors \mathbf{a}' , \mathbf{b}' , \mathbf{c}' and \mathbf{d} with

$$\mathbf{a}'_k = \mathbf{a}_k/d_k, \quad \mathbf{b}'_k = \mathbf{b}_k/d_k, \quad \mathbf{c}'_k = \mathbf{c}_k/d_k, \quad (17)$$

as then the number of floating point operations in the sparse matrix-vector multiplication can be further reduced. To reconstruct the matrix stencil the coefficients $|T_{ij}|$ and $\alpha_{i'j'}$ are also needed (we write $\alpha_{ij} \equiv \alpha_{T_{ij}}$ and $\alpha_{i'j'} \equiv \alpha_{T_{ij}, T_{i'j'}}$ for $T_{i'j'} \in \mathcal{N}(T_{ij})$ for simplicity). While these can be given

by relatively complicated algebraic expressions on complex geometries, such as the spherical grid considered in this article, they only need to be calculated once in each vertical column and for large n_z this will only lead to a small overhead.

On the regular horizontal grid which we use in our implementation the sparse matrix-vector product $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$ can then be calculated in each grid cell (i, j, k) as

$$\begin{aligned} y_{ijk} \leftarrow & \left[((a'_k - b'_k - c'_k) \cdot |T_{ij}| - \alpha_{ij}) \cdot x_{ijk} \right. \\ & + |T_{ij}| \cdot b'_k \cdot x_{i,j,k+1} + |T_{ij}| \cdot c'_k \cdot x_{i,j,k-1} \\ & + \alpha_{i+1,j} \cdot x_{i+1,j,k} + \alpha_{i-1,j} \cdot x_{i-1,j,k} \\ & \left. + \alpha_{i,j+1} \cdot x_{i,j+1,k} + \alpha_{i,j-1} \cdot x_{i,j-1,k} \right] \cdot d_k \end{aligned} \quad (18)$$

(with possible modifications at the boundary of the domain). For each (i, j, k) this requires 20 floating point operations and 12 memory references (7 loads for \mathbf{x} , 1 store for \mathbf{y} and 4 loads for a'_k, b'_k, c'_k and d_k). However, as the vectors \mathbf{a}' , \mathbf{b}' , \mathbf{c}' and \mathbf{d} do not change from column to column these are likely to remain in cache, reducing the number of memory references to 8. Depending on the innermost loop two of the values of \mathbf{x} (namely the ones belonging to the same vertical column which are needed at the next vertical level, i.e. x_{ijk} and $x_{i,j,k+1}$) will most likely stay in cache, which reduces the number of memory references further to 6. In contrast 14 floating point operations and 22 memory references are necessary if the matrix \mathbf{A} is stored in compressed sparse row storage (CSR) format. Again, the actual number of memory references is likely to be smaller as some of the variables are cached. However, in this case caching is not possible for the matrix entries which vary from one three dimensional grid cell to the next.

For the construction of the preconditioner we drop the second term in (5), which couples different vertical columns, and write $\mathbf{A}_{T_{ij}} \bar{\mathbf{u}}^{(i,j)} = \bar{\mathbf{b}}^{(i,j)}$ (using (12) to implicitly convert between the global vector representation and the column based representation in (4)). For each column (i, j) the tridiagonal system $\mathbf{A}_{T_{ij}} \bar{\mathbf{x}}^{(i,j)} = \bar{\mathbf{y}}^{(i,j)}$ defined by

$$\begin{aligned} |T_{ij}| \cdot d_k \cdot \left[\bar{x}_{k-1}^{(i,j)} \cdot c'_k \right. \\ \left. + \bar{x}_k^{(i,j)} \cdot ((a'_k - b'_k - c'_k) - \tilde{\alpha}_{ij}) \right. \\ \left. + \bar{x}_{k+1}^{(i,j)} \cdot b'_k \right] = \bar{y}_k^{(i,j)} \end{aligned} \quad (19)$$

with $\tilde{\alpha}_{ij} \equiv \alpha_{ij}/|T_{ij}|$ needs to be solved for $\bar{\mathbf{x}}^{(i,j)}$. This can be done efficiently in $\mathcal{O}(n_z)$ time using the Thomas algorithm, which is essentially Gaussian elimination applied to a tridiagonal system. The forward iteration requires 8 memory references at each step (loading the auxilliary vector ϕ_{k-1} and $y_{k-1}^{(i,j)}$ and storing ϕ_k and $y_k^{(i,j)}$ plus four loads for a'_k, b'_k, c'_k and d_k). In each step of the backward iteration 4 memory references are needed. Hence the total number of memory references is 12. Again, some of the data might be kept in cache. If only the vectors \mathbf{a}' , \mathbf{b}' , \mathbf{c}' and \mathbf{d} are cached the

number of memory references reduces to 8. If in addition any data in the same vertical column can be kept in cache, the number of memory references reduces further to 5. As there are no dependencies in the horizontal direction, we can parallelise in this direction, i.e. the tridiagonal solve in each vertical column can be carried out independently.

An additional advantage of the matrix free method is the fact that there are no matrix setup costs; the cost for precomputing the vectors \mathbf{a}' , \mathbf{b}' , \mathbf{c}' , \mathbf{d} and copying them to the device is negligible as these vectors only have length n_z .

4.3 Interleaved PCG algorithm

Field vectors are accessed in different components of the PCG algorithm, for example the residual \mathbf{r} is needed in the preconditioner solve, in the update of the residual and the calculation of the residual norm. In the standard implementation of the algorithm these operations are carried out in separate loops over the grid, which increases the number of memory references as data can not be kept in cache. However, the main iteration of the PCG algorithm can be rewritten such that it only consists of two loops over the grid, each of which contains either the sparse matrix-vector multiplication or the tridiagonal solve and a number of BLAS operations. Similar loop fusion for the a GPU implementation of the PCG algorithm presented in [43] has been discussed in [20].

The main iteration of this modified algorithm is shown in Algorithm 1. The kernels are written down explicitly for

Algorithm 1 Interleaved PCG loop

- 1: **for** $k = 1, \text{maxiter}$ **do**
 - 2: Interleaved preconditioner kernel: Calculate $\mathbf{r} \leftarrow \mathbf{r} - \alpha \mathbf{q}$, $\mathbf{z} = \mathbf{M}^{-1} \mathbf{r}$, $\|\mathbf{r}\| \leftarrow \sqrt{\langle \mathbf{r}, \mathbf{r} \rangle}$, $\kappa \leftarrow \langle \mathbf{r}, \mathbf{z} \rangle$ in a single iteration over the grid.
 - 3: **if** $(\|\mathbf{r}\|/\|\mathbf{r}_0\| < \epsilon \text{ or } \|\mathbf{r}\| < \tau)$ **then** Exit
 - 4: $\beta \leftarrow \kappa/\kappa_{old}$, $\kappa_{old} \leftarrow \kappa$
 - 5: Interleaved SpMV kernel: Calculate $\mathbf{u} \leftarrow \mathbf{u} + \alpha \mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta \mathbf{p}$, $\mathbf{q} \leftarrow \mathbf{A} \mathbf{z} + \beta \mathbf{q}$, $\sigma \leftarrow \langle \mathbf{p}, \mathbf{q} \rangle$ in a single iteration over the grid.
 - 6: $\alpha \leftarrow \kappa_{old}/\sigma$
 - 7: **end for**
-

the matrix-free implementation in Appendix A. The number of floating point operations and memory references for the matrix-free PCG algorithm and for its interleaved version are shown in Tab. 2. For the memory references we give three different values corresponding to the following assumptions: (1) no data is kept in cache, (2) only the vectors \mathbf{a}' , \mathbf{b}' , \mathbf{c}' and \mathbf{d} are cached and (3) in addition data in the same vertical column is cached.

While the algorithm is still memory bound on the GPU, the number of memory references is reduced in the interleaved implementation, in particular if the cache can be used efficiently.

algo- rithm	operation	FLOPs	Memory references		
			no cache	matrix cached	columns cached
PCG	SpMV	20	12	8	6
	prec	13	12	8	5
	BLAS	13	16	16	16
	total	46	40	32	27
Inter- leaved PCG	SpMV	28	17	13	11
	prec	19	16	12	9
	total	47	33	25	20

Table 2: Number of floating point operations and memory references per iteration and per grid point for different components of the PCG and the interleaved PCG algorithm. Memory references are shown without caching, with caching the vectors \mathbf{a}' , \mathbf{b}' , \mathbf{c}' and \mathbf{d} only and assuming that all degrees of freedom in a vertical column are cached as well.

5 Graphics Processing Units

In contrast to CPUs, on which most transistors are used for advanced execution control units and cache hierarchies, GPUs have a very large number (several hundreds to thousands) of lightweight compute cores which support SIMD parallelism for simple compute kernels and are ideally suited for floating point intensive calculations on regular data. The clock speed of each individual core is smaller than on the CPU, which improves the power efficiency.

The cores in the GPU are grouped into a number of streaming multiprocessors (SMs). Data can be stored in global on-chip GPU memory and in addition, each SM has a smaller and faster shared memory. Each compute core can also access an even smaller local memory in addition to a set of registers. Constants can be stored in fast constant memory. Modern GPUs, such as the Fermi architecture (see [44]), also have a hardware managed L1/L2 cache hierarchy. Data transfer between host and device memory has to be managed explicitly by the user. As typically the PCIe bus has a small bandwidth (at the order of ten GB/s), this is expensive and data should be calculated and kept on the GPU as long as possible.

5.1 CUDA programming model

The CUDA programming model described in the CUDA programming guide ([45]) provides an extension of the C language. When writing code for a GPU, the most compute intensive subroutines are parallelised by isolating them in simple kernels. On the GPU these kernels are executed by lightweight threads which are run on the compute cores of the SMs. Threads are grouped into blocks in a one-, two- or three dimensional grid, which execute independently on one SM. Synchronisation and data exchange is only possible between threads in the same block. In each block the threads are arranged into a grid, i.e. each thread is uniquely

identified by its thread index and block index. Threads are scheduled in groups of size 32, called warps. The threads in each warp execute one common instruction at a time; if the execution path diverges, for example due to an if-statement in the kernel, both branches are executed. This is known as thread divergence and should be avoided whenever possible.

As our solver algorithm is memory bound, performance can be increased by minimising latency and increasing global memory bandwidth. If a warp stalls as it is waiting for data from memory, it is paused and the next warp which is ready for execution is launched by the warp scheduler. In contrast to threads on a multicore CPU, the GPU scheduler is designed for launching and switching threads with minimal overheads. Thus, given the number of threads is large enough, memory latency can be hidden. To achieve this the GPU usually has to be oversubscribed, i.e. the number of threads launched at a single time should exceed the number of compute cores.

For optimal efficiency, memory access for all threads in one half-warp should be coalesced. Global memory access is processed in segments of 128 bytes (which is the size of one cache line) and all threads in a half-warp should access the smallest possible number of different segments. For example, if each of the 16 threads reads a double precision (8 byte) floating point number from global memory, this will require the transfer of one segment if these numbers are stored consecutively, but it will require 16 separate memory transfers and thus incur a big penalty if the numbers are more than 128 byte apart in global memory.

6 CUDA implementation of the PCG algorithm

Both the standard PCG method and its interleaved version were implemented in C and CUDA-C. In the standard version the sparse matrix-vector multiplication and preconditioner were implemented as kernels and the CUBLAS library was used for the level 1 BLAS operations.

To minimise memory transfers between host and device, data is kept on the device inside the entire PCG loop and all operations are carried out on the GPU. Host-device data transfers are only necessary for copying the initial solution and the right hand side to GPU memory before the CG iteration and for copying the final solution back to the host at the end. In addition, the four vectors \mathbf{a}' , \mathbf{b}' , \mathbf{c}' and \mathbf{d} describing the vertical discretisation need to be copied to the device once before the main PCG iteration. However, as each vector only has length n_z , the amount of data transferred is negligible in comparison to the size of the initial solution and right hand side vector. In particular it is significantly less than for matrix-explicit implementations.

6.1 Domain decomposition and data layout

Due to the inherently sequential nature of the Thomas algorithm, parallelisation in the vertical direction is not possible for the preconditioner. Instead the code is parallelised by assigning each vertical column to one thread and organising these into threadblocks of size $B = B_x \times B_y$, each of which is launched on one streaming multiprocessor. Parallelisation only in the horizontal direction is common in atmospheric models where data dependencies in the vertical direction are introduced by physical processes such as radiative transfer. To achieve a good occupancy the number of threads per block should be large, in particular latency hiding can only occur for $B \gg 32$.

In the code all three dimensional field vectors, such as the solution vector \mathbf{u} , are represented as one dimensional arrays of size $n = m \times m \times n_z$ which are stored contiguously in memory. To access the entry u_{ijk} the three dimensional index $(i, j, k) \in [0, m-1] \times [0, m-1] \times [0, n_z-1]$ (where k is the vertical index) has to be mapped to a linear index $\ell \in \{0, \dots, n-1\}$. To achieve optimal cache usage on the CPU, vertical columns are stored consecutively in memory in the C code, which can be achieved by the mapping

$$\ell^{(C)} = n_z \cdot (m \cdot i + j) + k \quad (20)$$

already introduced in (12). However, on a GPU, the data layout in (20) would lead to a significant amount of uncoalesced memory access as the number of vertical levels is large. For a typical n_z of 128, consecutive threads will access data which is $128 \times \text{sizeof}(\text{float})$ bytes apart in memory. Although this problem can be mitigated by use of the L1 cache (or manual prefetching into shared memory at the beginning of each kernel), on a GPU the L1 cache is shared between a large number of threads, which severely limits the cache size per thread. As each thread processes an entire vertical column, efficient caching would only be possible for relatively small horizontal block sizes B : in the Thomas algorithm two vectors of length n_z need to be stored per column in addition to the four vectors \mathbf{a}' , \mathbf{b}' , \mathbf{c}' and \mathbf{d} which describe the vertical discretisation. The total amount of L1 cache on the Fermi architecture is limited to 48kB, hence for single precision floating point numbers one would have

$$48 \text{ kB} \geq (2B + 4)n_z \times \text{sizeof}(\text{float}) \quad (21)$$

which would limit the number of threads per block to 44 for single precision and 22 for double precision calculations if we assume that $n_z = 128$.

An alternative approach is to change the storage format of the fields. Instead of (20) we use

$$\ell^{(\text{CUDA-C})} = m \cdot (n_z \cdot j + k) + i \quad (22)$$

in the matrix-free CUDA-C code, i.e. the first horizontal index runs fastest. This ensures that, provided the horizontal block size B_x is larger than 16, memory access for all threads in a half-warp is coalesced, as illustrated in Fig. 3.

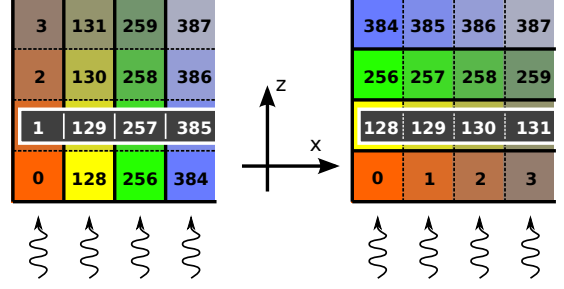


Figure 3: Memory layout and data access pattern. Data read by all threads in a warp is shown with a gray background. If the vertically continuous ordering of the degrees of freedom in (20) is used, memory access is not coalesced between the threads in one warp (left). The ordering degrees of freedom in (22) leads to coalesced memory access between all threads in a warp (right).

In our numerical tests we found that the blocksize $B_x = 64$, $B_y = 2$ gave good results, in particular the global load efficiency was almost 100% for the interleaved preconditioner and larger than 88% for the SpMV kernel. For this blocksize the data processed by one block is too large to fit into cache. However, we find that the L1 cache hit rate ranges between 33% for the interleaved preconditioner and 56% for the interleaved SpMV kernel. The total global memory bandwidth is around 25% – 50% of the peak value for both kernels. To reduce the number of cache misses further, a more fine grained parallelisation would have to be used to reduce the data volume per thread and ensure that data used by each thread block can be kept in shared memory.

The approach we are currently exploring (but which is not used in the implementation described in this article) is to use a different solver for the tridiagonal system in the vertical direction. The substructuring method discussed in [46] splits the $n_z \times n_z$ problem into B_z smaller tridiagonal systems of size $\approx (n_z/B_z) \times (n_z/B_z)$, which can be solved independently by different threads. To obtain the global solution on the entire vector of length n_z , a global tridiagonal system of size $B_z \times B_z$ for the interface points has to be solved before the solutions of the small subsystems can be combined to the total solution.

6.2 Matrix-explicit implementation

In addition to the matrix-free implementation described in section 4.2 we also wrote a version of the code based on an explicit representation of the matrix. Because of its popularity in the literature on sparse matrix-vector products and Krylov-subspace solvers (see the discussion in section 2) we chose the CSR format. The $n \times n$ tridiagonal matrix used in the preconditioner was stored as a set of three vectors of length n . The `cusparses[S]Dcsrsv()` function from the CUSPARSE library was used for the sparse matrix-vector multiplication and the subroutine `cusparses[S]Dgtsv()`

for the tridiagonal solve in the preconditioner.

Setup of the matrix on the host and copying the CSR representation to the device would create additional costs as host-device data transfers are expensive. For a fair comparison the matrices were set up on the device instead. Our numerical tests show that in this case the setup costs only form a small part of the total runtime. However, this might not be the case in other applications where matrix has to be constructed on the CPU.

In addition we wrote a CPU version using our own hand-written CSR matrix-vector product and the LAPACK routines GTTRF, GTTRS for the tridiagonal solver.

7 Numerical experiments

7.1 Hardware and compilers

All runs were carried out on the GPU node of the *aquila* cluster in Bath. The node contains an Intel Xeon E5-2620 Sandybridge CPU with a clockspeed of 2.00GHz and an nVidia Fermi M2090 GPU. The theoretical peak performance of one core of the Sandybridge CPU without AVX extensions is 8.0GFLOPs (4 floating point operations per cycle \times 2.0 GHz). The M2090 GPU contains 512 cores running at a clockspeed of 1.3GHz which are organised into 16 streaming multiprocessors with 32 cores each, as described in the Fermi Architecture Whitepaper ([44]). The total size of global GPU memory is 6GB, and each SM has 64kB of on-chip memory which can be split between shared memory and the L1 cache as 48kB/16kB or 16kB/48kB. In our implementation we used API calls to choose the optimal partitioning. In addition the Fermi architecture has 768kB of global L2 cache. The theoretical peak performance is quoted as 1.331TFLOPs for single precision and 0.665 TFLOPs for double precision while the peak bandwidth for access to global memory is quoted as 177GB/sec. Dividing the peak performance by the peak bandwidth implies that around 30 floating point operations can be carried out for each single/double precision variable read from global memory. In practise the actual number of floating point operations per accessed variable will of course be different due to latency effects and as required data might already be in L1 cache. However, this again stresses the importance of optimising memory throughput to achieve good performance.

In contrast, on the CPU the theoretical peak bandwidth is 41.6GB/s, and the number of floating point operations per value loaded from memory is 0.76 for single- and 1.54 for double precision.

The nVidia *nvcc* compiler (release 5.0, V0.2.1221) was used for compiling the CUDA code and we used version 4.4.6 of the gnu C compiler for compilation of the CPU code. To achieve the best possible performance of the matrix-explicit CPU code, optimised BLAS and LAPACK libraries based on the OpenBLAS implementation (see [47]) were used. AVX extensions were disabled on the CPU.

matrix-explicit kernel	time per call		speedup C vs. CUDA
	C	CUDA	
<i>single precision</i>			
SpMV	170.6	6.91	25 \times
preconditioner	205.3	12.50	16 \times
<i>double precision</i>			
SpMV	182.2	10.91	17 \times
preconditioner	249.7	23.20	11 \times

Table 3: Measured times and speedups for one sparse matrix-vector multiplication (SpMV) $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$ and one preconditioner solve $\mathbf{x} \leftarrow \mathbf{M}^{-1}\mathbf{y}$ on the CPU and GPU using the matrix-explicit implementation. All times are given in milliseconds. The speedup of the CUDA-C code relative to the sequential CPU implementation is shown in the last column for each case.

7.2 Results

We first study the performance and speedup of the individual components of the PCG solver and of the entire algorithm for a fixed problem of size $256 \times 256 \times 128$ with $\omega^2 = 6.71 \cdot 10^{-4}$ and $\lambda^2 = 3.32 \cdot 10^{-2}$, which is a typical set of parameters in meteorological applications. On the GPU we used a two-dimensional block layout and each block has a size of $B_x \times B_y = 64 \times 2$ threads. We found that this gives good results and varying the block size did not increase the performance.

7.3 Matrix-vector multiplication and preconditioner

In Tab. 3 the times for a single sparse matrix-vector multiplication and preconditioner solve are shown for the matrix-explicit method. These times do not include any costs for setting up the matrix or for transferring data between host and device, as this is only required once for each PCG solve, which consists of a large number of sparse matrix-vector multiplications and preconditioner applications. The speedups shown in this table are relative to the sequential CPU implementation. Assuming that the CPU code can be parallelised perfectly, the socket-to-socket speedup is a factor 6 smaller as the CPU contains six processor cores.

The corresponding times for the matrix-free implementation are shown in Tab. 4, where we also show the speedup of the matrix-free CUDA-C code relative to the matrix-explicit CUDA-C code. On the CPU the matrix-free sparse matrix-vector multiplication is more than twice as fast as the matrix-explicit implementation. However, the matrix-explicit preconditioner is 25% – 30% faster than the corresponding matrix-free version. On the GPU the matrix-free code is significantly faster than the matrix-explicit version, both for the sparse matrix-vector multiplication where the speedup is 9.2 \times for single precision and 7.7 \times for double precision. For the preconditioner the speedup is slightly smaller

matrix-free kernel	time per call		speedup	
	C	CUDA	C vs. CUDA	mat.-free vs. CSR
<i>single precision</i>				
SpMV	78.5	0.75	105×	9.2×
preconditioner	252.6	2.40	105×	5.2×
interlvd. SpMV	129.9	2.16	60×	—
interlvd. prec.	253.1	3.34	76×	—
<i>double precision</i>				
SpMV	80.7	1.41	57×	7.7×
preconditioner	350.4	3.75	93×	6.2×
interlvd. SpMV	132.3	3.86	34×	—
interlvd. prec.	351.3	4.86	72×	—

Table 4: Measured times and speedups for one sparse matrix-vector multiplication (SpMV) $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$ and preconditioner solve $\mathbf{y} \leftarrow \mathbf{M}^{-1}\mathbf{x}$ on the CPU and GPU using the matrix-free implementation. All times are given in milliseconds. The last two columns show the speedup of the matrix-free CUDA-C code relative to the corresponding sequential CPU implementation and relative to the matrix-explicit CUDA-C version.

with $5.2\times$ and $6.2\times$ for single- and double precision respectively. The speedup of both standalone matrix-free GPU kernels relative to the sequential CPU code is more than $100\times$ in single precision. In double precision the speedup is $93\times$ for the preconditioner and $57\times$ for the sparse matrix-vector multiplication. While the corresponding speedups for the interleaved kernels are smaller, their performance has to be judged in the context of the full PCG loop, which for the standalone kernels also contains several level 1 BLAS operations.

We observe that for the matrix-free standalone SpMV kernel the double precision implementation takes about twice as long as the single precision version. This suggests that the implementation is bandwidth limited and less affected by cache efficiency, as the double precision version requires transferring twice as much data from global memory than the single precision implementation. This interpretation is also corroborated by the relatively high cache hit rate for this kernel reported in Tab. 7. The cache hit rate is smaller for the interleaved sparse matrix-vector multiplication and the preconditioner kernels, all of which show a smaller increase in the runtime between single and double precision. A similar drop of performance by nearly a factor of two when going from single to double precision can be observed for the matrix-explicit kernels, see Tab. 3.

7.4 PCG algorithm

We now analyse the performance of the entire PCG solver and break down the time spent in a single iteration of the algorithm.

7.4.1 Total solution time

We measured the time required to carry out in 100 PCG iterations, which is sufficient to reduce the residual by five orders or magnitude. Our measurements include the time for the matrix setup and data transfer between host and device. The results are listed for three different implementations of the algorithm in Tab. 5 where we also calculated the speedup of the matrix-free CUDA-C code both relative to the C code and relative to the matrix-explicit GPU code. The matrix-free interleaved algorithm gives the best performance, with a speedup of a factor of $60\times$ (single precision) and $48\times$ (double precision) relative to the C code on the CPU. It outperforms the matrix-explicit GPU code by more than a factor of four. On the CPU the interleaved algorithm is only slightly faster than standard PCG for single precision and even slightly slower for double precision. This is because in the sequential implementation most of the time ($80\% - 90\%$) is spent in the sparse matrix-vector multiplication and preconditioner kernels, so fusing the kernels can only give a speedup of no more than $10\% - 20\%$. This is different on the GPU, as will be discussed in section 7.4.2.

As expected the cost for setting up the vertical discretisation matrix (calculation of \mathbf{a}' , \mathbf{b}' , \mathbf{c}' and \mathbf{d}) and copying it to the device turned out to be negligible ($\ll 1ms$). For the matrix-explicit code the matrix setup time only accounts for a small proportion of the runtime; on the GPU the matrix setup time is 6% and 4% of the total solution time in single- and double precision respectively. Although for the matrix-free interleaved code host-device data transfer of the solution and right hand side vector takes up only a small part of the runtime (about 8% both in single- and double precision), this is not true any longer if a smaller number of iterations is used for example to only reduce the residual by three orders of magnitude. We also found that on the CPU the total solution time can be reduced by a factor of around four if the Krylov- subspace solver is replaced by a geometric solver multigrid, as is discussed in [36]. Again, this would increase the relative importance of the host-device memory transfer.

7.4.2 Time per iteration

The time per iteration is given for the three different implementations of the PCG algorithm in Tab. 6 where we also calculate the speedup of the matrix-free implementation relative to the matrix-explicit version. The speedup relative to the C implementation is $66\times$ for the matrix-free interleaved implementation in single precision and $52\times$ in double precision. In both cases it is more than four times faster than the matrix-explicit implementation on the GPU. Comparing the total time per iteration for the matrix-free and the interleaved implementation, we find that the ratio between the two is 0.69 in single precision and 0.63 in double precision, which should be compared to the corresponding ratio of memory references in Tab. 2, which is $33/40 = 0.83$ without caching or $20/27 = 0.74$ assuming that the vectors

implementation	matrix and preconditioner setup		data transfer CUDA	total time		speedup	
	C	CUDA		C	CUDA	C vs. CUDA	matrix-free vs. CSR
<i>single precision</i>							
matrix-explicit	0.55	0.15	0.047	43.72	2.54	17×	—
matrix-free	—	—	0.047	37.00	0.87	43×	2.9×
matrix-free interleaved	—	—	0.047	37.39	0.62	60×	4.1×
<i>double precision</i>							
matrix-explicit	0.72	0.16	0.073	50.50	4.41	11×	—
matrix-free	—	—	0.073	49.37	1.48	33×	3.0×
matrix-free interleaved	—	—	0.073	45.78	0.96	48×	4.6×

Table 5: Total solution time for different implementations of the PCG algorithm. Costs for matrix setup and host-device data transfer are listed separately and included in the total times. 100 iterations of the PCG main loop were carried out in all cases and all times are given in seconds.

implementation	time per iteration		speedup	
	C	CUDA	C vs. CUDA	mat.-free vs. CSR
<i>single precision</i>				
matrix-explicit	410.8	23.5	17×	—
matrix-free	376.3	8.1	46×	2.9×
—"—" interlvd.	370.6	5.6	66×	4.2×
<i>double precision</i>				
matrix-explicit	494.6	41.4	12×	—
matrix-free	491.5	13.9	35×	3.0×
—"—" interlvd.	453.4	8.8	52×	4.7×

Table 6: Time per iteration for different implementations of the conjugate gradient algorithm. All times are measured in milliseconds.

\mathbf{a}' , \mathbf{b}' , \mathbf{c}' and \mathbf{d} and all data in one vertical column is cached (the ratio is $25/32 = 0.78$ if only the \mathbf{a}' , \mathbf{b}' , \mathbf{c}' and \mathbf{d} are cached). To identify the remaining bottlenecks, these times are further broken down for the GPU code in Figs. 4 and 5 for single- and double precision arithmetic. Note that for the matrix-free code the BLAS operations, and in particular the `axpy` updates, make up a significant proportion of the time in the matrix-free code. The plot clearly shows the benefit of the interleaved implementation, which increases the performance by 28% in single precision and 35% in double precision.

7.5 Absolute performance

While comparing the runtime of the CUDA-C code to the corresponding sequential CPU implementation can give an idea of the achievable performance gains, it is somewhat arbitrary in that it depends on the exact CPU which is used for this comparison. For this reason we also quantified the absolute performance of the matrix-free CUDA-C code.

Several performance indicators for the kernels of the matrix-free code are shown in Tab. 7. The global load

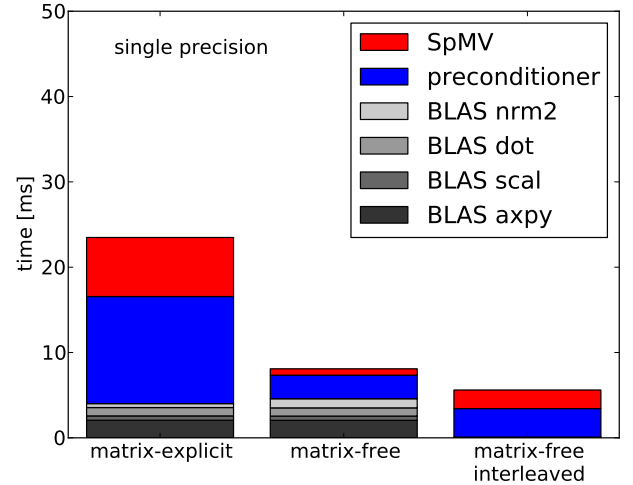


Figure 4: Time per iteration for different parts of the main CG loop on the device using single precision. The BLAS-operations were implemented with the CUBLAS library.

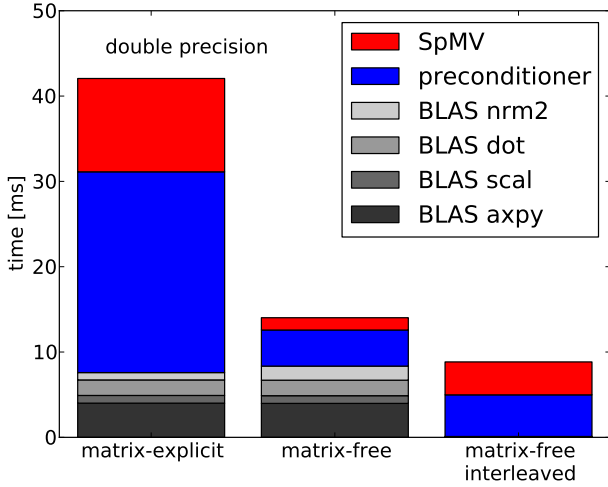


Figure 5: Time per iteration for different parts of the main CG loop on the device using double precision. The BLAS-operations were implemented with the CUBLAS library.

efficiency measures the amount of coalesced global memory access and the L1 hit rate quantifies the cache efficiency. For all kernels load efficiency is very high due to coalescence of global memory access as described in section 6.1.

The floating point performance for one iteration of the matrix-free interleaved PCG algorithm is plotted for a range of problem sizes between $2.1 \cdot 10^6$ and $1.3 \cdot 10^8$ in Fig. 6. The nVidia M2090 Fermi GPU has a global memory of 6GB, and as the PCG algorithm requires the storage of 5 field vectors, which limits the problem size to less than $3 \cdot 10^8$ for single precision and $1.5 \cdot 10^8$ for double precision. In all cases the size of a vertical column was kept fixed at $n_z = 128$. The performance is virtually independent of the problem size and about 70-80 GFLOPs for single- and 40-50 GFLOPs for double precision. As the algorithm is memory bound, a more relevant measure is the global memory band width which is shown for the interleaved kernels in Fig. 7. The bandwidth increases slightly with the problem size and 25% – 50% of the peak value could be achieved.

8 Conclusions

In this article we described the matrix-free implementation of a Preconditioned Conjugate Gradient solver for strongly anisotropic elliptic PDEs on a GPU. Equations of this type arise in many applications in atmospheric and geophysical modelling if the problem is discretised in “flat” geometries. In particular the semi-implicit semi-Lagrangian time discretisation of the non-hydrostatic Euler equations in the dynamical core of many weather- and climate prediction models leads to a three dimensional PDE for the pressure correction and due to the small thickness of the atmosphere the elliptic operator has a very strong anisotropy in the vertical direction. This anisotropy can be exploited to construct a

	GFLOPs	memory bandwidth [GB/s]	global load ef- ficiency	L1 hit rate
<i>single precision</i>				
SpMV	223.7	50.73	86.2%	76.7%
preconditioner	45.4	38.41	99.9%	40.4%
interlvd. SpMV	108.7	64.67	88.8%	56.5%
interlvd. prec.	47.7	46.16	99.6%	33.6%
<i>double precision</i>				
SpMV	119.0	79.07	85%	74.0%
preconditioner	29.1	49.35	99.7%	40.1%
interlvd. SpMV	60.9	78.97	88.0%	57.5%
interlvd. prec.	32.8	64.58	99.8%	33.1%

Table 7: Performance measurements of the different matrix-free kernels as reported by the nVidia visual profiler. The bandwidth is the DRAM load bandwidth.

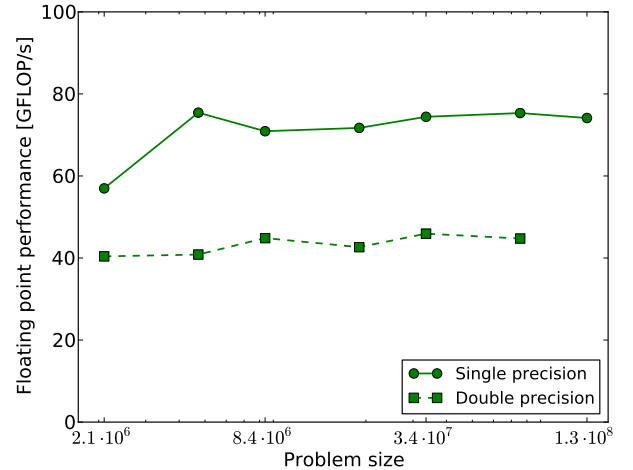


Figure 6: Floating point performance for different problem sizes for the matrix-free interleaved PCG code on the GPU.

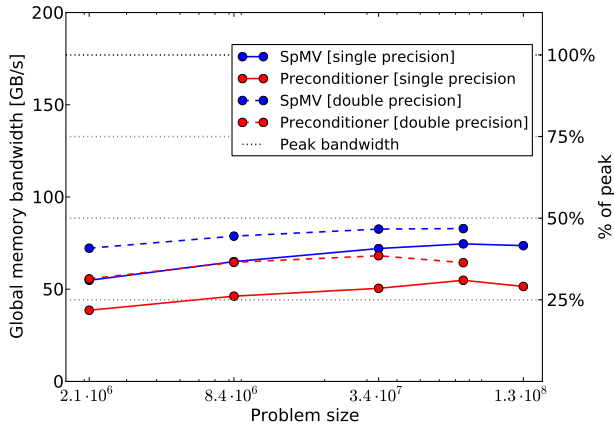


Figure 7: Global memory bandwidth (DRAM read bandwidth as reported by the nVidia Visual Profiler) for different problem sizes for the interleaved apply- and preconditioner kernels.

simple and efficient preconditioner based on vertical line relaxation. The dependencies in each vertical column require a horizontal domain decomposition, which has implementations for the data- and thread layout on the GPU.

As the performance of the algorithm is limited by the speed with which data can be transferred from global memory to the compute units, it is important to reduce the number of memory references. We achieved this by using a matrix-free implementation which recalculates the local matrix stencil whenever it is needed instead of reading it from memory. In addition, we reduced the amount of data transfer by fusing several kernels in the PCG loop, which gave an additional improvement of 28% in single precision and 35% in double precision. In total we demonstrated that on an nVidia Fermi M2090 GPU the best matrix-free code achieved a speedup of 60× in single precision and 48× for double precision, compared to the corresponding sequential implementation on an Intel Xeon E5-2620 Sandybridge CPU. In terms of absolute times, the residual for a problem with $8.3 \cdot 10^6$ degrees of freedom could be reduced by more than five orders of magnitude in 0.62 seconds in single precision. The double precision implementation takes about 1.5 times longer, which demonstrates the good double precision performance of modern GPUs. Our matrix-free version is more than four times faster than a matrix-explicit GPU implementation based on the CUSPARSE and CUBLAS libraries using the CSR matrix format which clearly demonstrates the benefit of our approach. We measured the absolute performance of our code for a range of problem sizes and achieved a global memory bandwidth of 25% – 50% of the peak rate for problem sizes between $2.1 \cdot 10^6$ and $1.3 \cdot 10^8$ degrees of freedom.

Global memory access was coalesced for the threads within a half warp by numbering the degrees of freedom such that the horizontal index runs fastest. This is different from CPU implementations where vertical columns are

stored consecutively in memory for cache efficiency. While the specific implementation discussed in this article is based on a regular horizontal grid, our method can be applied to any three dimensional grid which can be written as the tensor product of possibly unstructured horizontal grid and a non-uniform one dimensional grid in the vertical direction.

The achieved global memory bandwidth is a sizeable fraction of the peak value, and it can be theoretically improved by an additional factor 2× to 4× by making better use of the GPU cache or shared memory. This would require the parallelisation of the tridiagonal solver in the vertical direction, and we are currently investigating the substructuring approach described in [46].

With the planned increase in weather- and climate model resolution, problems with more than 10^{10} degrees of freedom need to be solved. Clearly this is not possible on a single GPU due to limited global memory size. To solve problems of this size hundreds of GPUs are necessary as each has limited memory. We are currently extending the algorithm to multi-GPU clusters will introduce additional bottlenecks: unless data can be copied directly between GPU memory, it has to be transferred to the host at each iteration before it can be sent through the standard MPI network. However, in this case only halo data needs to be exchanged between neighbouring devices and given that the local problem size is not too small, this is significantly less than the total data processed by one GPU.

The PCG solver described in this work requires around hundred iterations to reduce the residual by five orders of magnitude. In contrast, multigrid solvers can achieve the same reduction in a much smaller number of iterations, as has been demonstrated in [36] for the elliptic PDE considered in this article. The preconditioner used in this work can be used as a multigrid smoother and the only missing components are intergrid operators for restriction and prolongation, which is the object of our current research.

Acknowledgements

We would like to thank all members of the GungHo! project and in particular Chris Maynard and David Ham for useful and inspiring discussions. The numerical experiments in this work were carried out on a node of the aquila supercomputer at the University of Bath and we are grateful to Steven Chapman for his continuous and tireless technical support which was essential for the success of this project. The contribution of EM and RS was funded as part of the NERC project on “Next Generation Weather and Climate Prediction” (NGWCP), grant number NE/J005576/1.

A Interleaved PCG kernels

The kernels for the interleaved PCG algorithm described in section 4.3 are shown in Algorithms 2 and 3. In the GPU implementation each thread calculates dot products and norms

in one column. To obtain global sums, these need to be reduced with an additional BLAS operation. However, as this operation only operates on two-dimensional (horizontal) vectors, its cost is very small ($< 1\%$ of the time per iteration).

Algorithm 2 Interleaved matrix-multiplication kernel. Simultaneously calculate $\mathbf{u} \leftarrow \mathbf{u} + \alpha \mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta \mathbf{p}$, $\mathbf{q} \leftarrow \mathbf{A}\mathbf{z} + \beta \mathbf{q}$, $\sigma \leftarrow \langle \mathbf{p}, \mathbf{q} \rangle$ in a single iteration over the grid.

```

1: for  $i = 0, \dots, m$  do
2:   for  $j = 0, \dots, m$  do
3:     Calculate  $\alpha_{i',j'}$  and  $|T_{ij}|$ 
4:      $\sigma \leftarrow 0$ 
5:     for  $k = 0, \dots, n_z - 1$  do
6:        $p^* \leftarrow p_{ijk}$ ,  $q^* \leftarrow q_{ijk}$ ,  $z^* \leftarrow z_{ijk}$ 
7:        $u_{ijk} \leftarrow u_{ijk} + \alpha \cdot p^*$ 
8:        $p^* \leftarrow \beta \cdot p^* + z^*$ ,  $q^* \leftarrow \beta \cdot q^*$ 
9:        $p_{ijk} \leftarrow p^*$ 
10:       $\delta q \leftarrow ((a'_k - b'_k - c'_k) \cdot |T_{ij}| - \alpha_{ij}) \cdot z^*$ 
         $+ b'_k \cdot |T_{ij}| \cdot z_{i,j,k+1} + c'_k \cdot |T_{ij}| \cdot z_{i,j,k-1}$ 
         $+ \alpha_{i+1,j} \cdot z_{i+1,j,k} + \alpha_{i-1,j} \cdot z_{i-1,j,k}$ 
         $+ \alpha_{i,j+1} \cdot z_{i,j+1,k} + \alpha_{i,j-1} \cdot z_{i,j-1,k}$ 
11:       $q^* \leftarrow q^* + d_k \cdot \delta q$ ,  $\sigma \leftarrow \sigma + p^* \cdot q^*$ 
12:       $q_{ijk} \leftarrow q^*$ 
13:    end for
14:  end for
15: end for

```

References

- [1] M. Kwizak and A. J. Robert. a Semi-Implicit Scheme for Grid Point Atmospheric Models of the Primitive Equations. *Monthly Weather Review*, 99:32, 1971.
- [2] Andr   Robert. A stable numerical integration scheme for the primitive meteorological equations. *Atmosphere-Ocean*, 19(1):35–46, 1981.
- [3] P K Smolarkiewicz and L G Margolin. On forward-in-time differencing in fluids: An Eulerian/semi-Lagrangian nonhydrostatic model for stratified flows. *Atmosphere-Ocean*, special vol. XXXV(1):127–152, 1997.
- [4] S J Thomas, A V Malevsky, M Desgagne, R Benoit, P Pellerin, and M Valin. Massively Parallel Implementation of the Mesoscale Compressible Community Model. *Span*, pages 1–19, 1997.
- [5] W C Skamarock, P K Smolarkiewicz, and J B Klemp. Preconditioned conjugate-residual solvers for Helmholtz equations in nonhydrostatic models. *Monthly Weather Review*, 125(4):587–599, April 1997.
- [6] T. Davies, M. J. P. Cullen, A. J. Malcolm, M. H. Mawson, A. Staniforth, A. A. White, and N. Wood. A new dynamical core for the Met Office’s global and regional modelling of the atmosphere. *Quarterly Journal of*

Algorithm 3 Interleaved preconditioner kernel. Simultaneously calculate $\mathbf{r} \leftarrow \mathbf{r} - \alpha \mathbf{q}$, $R \equiv \|\mathbf{r}\| \leftarrow \sqrt{\langle \mathbf{r}, \mathbf{r} \rangle}$, $\kappa \leftarrow \langle \mathbf{r}, \mathbf{z} \rangle$ and solve $\mathbf{M}\mathbf{z} = \mathbf{r}$ in a single iteration over the grid.

```

1: for  $i = 0, \dots, m$  do
2:   for  $j = 0, \dots, m$  do
3:     Calculate  $\alpha_{i',j'}$  and  $|T_{ij}|$ 
4:      $R \leftarrow 0$ ,  $\kappa \leftarrow 0$ ,  $D \leftarrow (a'_k - b'_k - c'_k) - \alpha'_{ij}$ ,  $\phi_0 \leftarrow b'_k/D$ 
5:      $r^* \leftarrow r_{ij0} - \alpha \cdot q_{ij0}$ ,  $R \leftarrow R + r^* \cdot r^*$ 
6:      $z_{ij0} \leftarrow r^*/(D \cdot |T_{ij}| \cdot d_k)$ ,  $r_{ij0} \leftarrow r^*$ 
7:     for  $k = 0, \dots, n_z - 1$  do
8:        $D \leftarrow ((a'_k - b'_k - c'_k) - \alpha'_{ij}) - \phi_{k-1} \cdot c'_k$ ,  $\phi_k \leftarrow b'_k/D$ 
9:        $r^* \leftarrow r_{ijk} - \alpha \cdot q_{ijk}$ ,  $R \leftarrow R + r^* \cdot r^*$ 
10:       $z_{ijk} \leftarrow (r^*/(|T_{ij}| \cdot d_k) - c'_k \cdot z_{i,j,k-1})/D$ ,  $r_{ijk} \leftarrow r^*$ 
11:    end for
12:     $\kappa \leftarrow \kappa + z_{i,j,n_z-1} \cdot r_{i,j,n_z-1}$ 
13:    for  $k = n_z - 2, \dots, 0$  do
14:       $z^* \leftarrow z_{ijk} - \phi_k \cdot z_{i,j,k+1}$ ,  $\kappa \leftarrow \kappa + z^* \cdot r_{ijk}$ 
15:       $z_{ijk} \leftarrow z^*$ 
16:    end for
17:  end for
18: end for
19:  $R \leftarrow \sqrt{R}$ 

```

the Royal Meteorological Society, 131(608):1759–1782, April 2005.

- [7] Thomas Melvin, Mark Dubal, Nigel Wood, Andrew Staniforth, and Mohamed Zerroukat. An inherently mass-conserving iterative semi-implicit semi-Lagrangian discretization of the non-hydrostatic vertical-slice equations. *Quarterly Journal of the Royal Meteorological Society*, 136(648):799–814, 2010.
- [8] Nigel Wood, Andrew Staniforth, Andy White, Tom Allen, Michail Diamantakis, Markus Gross, Thomas Melvin, Chris Smith, Simon Vosper, Mohamed Zerroukat, and John Thuburn. An inherently mass-conserving semi-implicit semi-Lagrangian discretisation of the deep-atmosphere global nonhydrostatic equations. *submitted to Quarterly Journal of the Royal Meteorological Society*, 2013.
- [9] M R Hestenes and E Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, 1952.
- [10] W.L. Briggs, V.E. Henson, and S.F. McCormick. *A Multigrid Tutorial*. Society for Industrial and Applied Mathematics, 2000.
- [11] U Trottenberg, C W Oosterlee, and A Sch  ller. *Multigrid*. Academic Press, 2001.
- [12] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2 edition, April 2003.

- [13] John Marshall, Alistair Adcroft, Chris Hill, Lev Perelman, and Curt Heisey. A finite-volume, incompressible Navier Stokes model for studies of the ocean on parallel computers. *Journal of Geophysical Research*, 102:5753–5766, 1997.
- [14] O. B. Fringer and M. Gerritsen. An unstructured-grid, finite-volume, nonhydrostatic, parallel coastal ocean simulator. *Ocean Modelling*, 14:139–173, 2006.
- [15] S. Lacroix, Y. Vassilevski, J. Wheeler, and M. Wheeler. Iterative Solution Methods for Modeling Multiphase Flow in Porous Media Fully Implicitly. *SIAM Journal on Scientific Computing*, 25(3):905–926, 2003.
- [16] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Transactions on Graphics*, 22:917–924, 2003.
- [17] S Menon and JB Perot. Implementation of an efficient conjugate gradient algorithm for Poisson solutions on graphics processors. In *Proceedings of the 2007 Meeting of the Canadian CFD Society, Toronto Canada*, 2007.
- [18] R.F. Carvalho, C.A.P.S. Martins, R.M.S. Batalha, and A.F.P. Camargos. 3D parallel conjugate gradient solver optimized for GPUs. In *Electromagnetic Field Computation (CEFC), 2010, 14th Biennial IEEE Conference on*, page 1, may 2010.
- [19] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser. A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform. In *Parallel, Distributed and Network-Based Processing (PDP), 2010, 18th Euromicro International Conference on*, pages 583–592, feb. 2010.
- [20] M.M. Dehnavi, D.M. Fernández, and D. Giannacopoulos. Enhancing the Performance of Conjugate Gradient Solvers on Graphic Processing Units. *Magnetics, IEEE Transactions on*, 47(5):1162–1165, may 2011.
- [21] H. Knibbe, C.W. Oosterlee, and C. Vuik. GPU implementation of a Helmholtz Krylov solver preconditioned by a shifted Laplace multigrid method. *Journal of Computational and Applied Mathematics*, 236:281–293, 2011.
- [22] Martijn de Jong. Developing a CUDA solver for large sparse matrices for MARIN. Master’s thesis, Delft Institute of Applied Mathematics, 2012.
- [23] Rudi Helfenstein and Jonas Koko. Parallel preconditioned conjugate gradient algorithm on GPU. *Journal of Computational and Applied Mathematics*, 236(15):3584–3590, 2012. Proceedings of the Fifteenth International Congress on Computational and Applied Mathematics (ICCAM-2010), Leuven, Belgium, 5-9 July, 2010.
- [24] I. Reguly and M. Giles. Efficient sparse matrix-vector multiplication on cache-based GPUs. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12, may 2012.
- [25] Ruipeng Li and Yousef Saad. GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing*, 63:443–466, 2013.
- [26] Timothy A. Davis and Yifan Hu. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.
- [27] M.M. Dehnavi, D.M. Fernández, and D. Giannacopoulos. Finite-Element Sparse Matrix Vector Multiplication on Graphic Processing Units. *Magnetics, IEEE Transactions on*, 46(8):2982–2985, aug. 2010.
- [28] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC ’09*, pages 18:1–18:11, New York, NY, USA, 2009. ACM.
- [29] Dominik Michels. Sparse-matrix-cg-solver in cuda. In *Proceedings of the 15th Central European Seminar on Computer Graphics*, 2011.
- [30] Graham R. Markall, David A. Ham, and Paul H.J. Kelly. Towards generating optimised finite element solvers for GPUs from high-level specifications. *Procedia Computer Science*, 1(1):1815–1823, 2010. ICCS 2010.
- [31] C.D. Cantwell, S.J. Sherwin, R.M. Kirby, and P.H.J. Kelly. From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements. *Computers & Fluids*, 43(1):23–28, 2011. Symposium on High Accuracy Flow Simulations. Special Issue Dedicated to Prof. Michel Deville.
- [32] Nolan Goodnight, Cliff Woolley, Gregory Lewin, David Luebke, and Greg Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses, SIGGRAPH ’05*, New York, NY, USA, 2005. ACM.
- [33] Markus Geveler, Dirk Ribbrock, Dominik Göddeke, Peter Zajac, and Stefan Turek. *Efficient finite element geometric multigrid solvers for unstructured grids on GPUs*. Techn. Univ., Fak. für Mathematik, 2011.
- [34] James Brannick, Yao Chen, Xiaozhe Hu, and Ludmil Zikatanov. Parallel Unsmoothed Aggregation Algebraic Multigrid Algorithms on GPUs. *arXiv preprint arXiv:1302.2547*, 2013.
- [35] S. Börm and R. Hiptmair. Analysis Of Tensor Product Multigrid. *Numer. Algorithms*, 26:200–1, 1999.
- [36] Eike Müller and Robert Scheichl. Massively parallel solvers for elliptic PDEs in Numerical Weather- and Climate Prediction. *in preparation*, 2013.
- [37] Ali Cevahir, Akira Nukada, and Satoshi Matsuoka. *Fast Conjugate Gradients with Multiple GPUs*, volume 5544 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009. Editors: Allen, Gabrielle and Nabrzyski, Jarosław and Seidel, Edward and Albada, Geert Dick and Dongarra, Jack and Sloot, Peter M.A.

- [38] Serban Georgescu and Hiroshi Okuda. Conjugate gradients on multiple GPUs. *International Journal for Numerical Methods in Fluids*, 64(10-12):1254–1273, 2010.
- [39] Mickeal Verschoor and Andrei C. Jalba. Analysis and performance estimation of the Conjugate Gradient method on multiple GPUs. *Parallel Comput.*, 38(10-11):552–575, October 2012.
- [40] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007.
- [41] P K Smolarkiewicz and L G Margolin. Variational solver for elliptic problems in atmospheric flows. *Appl. Math. and Comp. Sci.*, 4:101–125, 1994.
- [42] Zbigniew Piotrowski, Andrzej Wyszogrodzki, and Piotr Smolarkiewicz. Towards petascale simulation of atmospheric circulations with soundproof equations. *Acta Geophysica*, pages 1–18, 2011.
- [43] A.T. Chronopoulos and C.W. Gear. s-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25(2):153 – 168, 1989.
- [44] nVidia Corporation. Fermi architecture whitepaper, 2009. Accessed 9 February 2013.
- [45] nVidia Corporation. CUDA programming guide, 2012. Accessed 9 February 2013.
- [46] A. Toselli and O.B. Widlund. *Domain Decomposition Methods - Algorithms and Theory*. Springer Series in Computational Mathematics. Springer, 2005.
- [47] Zaheer Chothia Zhang Xianyi, Wang Qian. OpenBLAS, 2012. Accessed 9 February 2013.